An Activity Selection Problem

- We are given n activities indexed from 1 to n
- Activity i has start time s_i and finish time f_i
- We cannot participate in two activities that overlap in time
- We wish to determine a maximum-cardinality set of non-overlapping activities

Key Observation

- Let i be an activity with minimum finish time
- Claim: Some optimal solution includes i
- To prove this claim, we can use an "exchange argument"
  - Suppose S is an optimal solution that does not include i
  - Let j be the first activity in S
  - Then $(S - j) + i$ is an optimal solution that includes i

A Greedy Algorithm

- Re-index the activities in nondecreasing order of finish time
- Initialize I to $\{1, \ldots, n\}$ and S to $\emptyset$
- While $I \neq \emptyset$
  - Let i be a minimum-index activity in I
  - Add i to S
  - Eliminate from I all indices j such that activities i and j overlap

A Fast Implementation

- Re-index the activities in nondecreasing order of finish time
- Initialize S to $\{1\}$ and k to 1
- For i running from 2 to n
  - If the start time of activity i is at least the finish time of activity k, then add i to S and set k to i

The Fractional Knapsack Problem

- Recall the knapsack problem
  - We are given a positive integer knapsack capacity W and n items indexed from 1 to n
  - Item i has positive integer value v_i and weight w_i
  - We wish to identify a maximum-value set of items with weight at most W
- In the fractional knapsack problem, we are allowed to take a fractional amount of any item

Key Observation

- Let i be an item with maximum "value density" v_i/w_i
- Claim: Some optimal solution includes a $z = \min(1, W/w_i)$ fraction of item i
- To prove this claim, we can use an exchange argument
  - Let S be an optimal solution that includes a fraction $z' < z$ of item i
  - Observe that the weight of S is at least z
  - Modify S by removing (fractional) items not equal to i with total weight $z - z'$, and replacing them with $z - z'$ units of item i
  - Observe that S remains optimal

A Greedy Algorithm

- Re-index the items in nonincreasing order of value density
- Take as much as possible of item 1, then as much as possible of item 2, et cetera, until the knapsack is full or there are no items left
- This algorithm uses $O(n \log n)$ operations due to the sorting (re-indexing) step
- Can we do better?

A Faster Implementation

- Recall the BFPRT linear-time selection algorithm
- It is easy to generalize the BFPRT algorithm to solve the following weighted selection problem in linear time
  - Each of the n keys in the input has a positive weight w_i
  - If some keys are equal, choose a way to break ties (e.g., by index) to obtain a total ordering of the keys
  - Let X denote $\sum_{1 \leq i \leq n} w_i$
  - We are given a desired threshold x, $0 \leq x \leq X$
  - We wish to identify the maximum key k such that the total weight of the keys preceding key k is less than x
- We can use a linear-time weighted selection algorithm to solve fractional knapsack in linear time

Scheduling to Minimize Maximum Lateness

- We are given n tasks indexed from 1 to n
- Task i has a positive integer deadline d_i and a positive integer execution requirement e_i
- We wish to (nonpreemptively) schedule all n tasks on a single resource beginning at time 0 in such a way that the maximum "lateness" of any task is minimized
  - A task with deadline d and termination time t is defined to have lateness $\max(0, t - d)$
- We can restrict attention to gap-free schedules, so we are optimizing over $n!$ schedules

Key Lemma

- Suppose S is a schedule in which task j is executed immediately after task i and $d_j \leq d_i$
  - Let ℓ_i (resp., ℓ_j) denote the lateness of task i (resp., j) in S
- Let S' be the schedule that is the same as S except that the order of execution of tasks i and j is interchanged
  - Let ℓ'_i (resp., ℓ'_j) denote the lateness of task i (resp., j) in S'
- Lemma: $\ell_j \geq \max(\ell'_i, \ell'_j)$
- This lemma implies that the "earliest deadline" rule yields an optimal schedule
  - Ties can be broken arbitrarily

Proof of the Key Lemma

- Lemma: $\ell_j \geq \max(\ell'_i, \ell'_j)$
  - Assume tasks i and j are executed in the time interval $[s, s + e_i + e_j]$ in S and S'
  - We have $\ell_j = \max(0, A)$ where $A = s + e_i + e_j - d_j$, $\ell'_i = \max(0, B)$ where $B = s + e_i + e_j - d_i$, and $\ell'_j = \max(0, C)$ where $C = s + e_j - d_j$
  - Observe that $A > C$
  - Since $d_j \leq d_i$, we have $A \geq B$
  - The lemma follows since

$$\ell_j = \max(0, A) \geq \max(0, B, C) = \max(\ell'_i, \ell'_j)$$

Single-Source Shortest Paths, Revisited

- Recall that we can solve the SSSP problem in $O(|E| \cdot |V|)$ using the Bellman-Ford algorithm, which can handle negative edge weights
- If the edge weights are nonnegative, we can solve the SSSP problem much more rapidly using Dijkstra's algorithm
- For any vertex v, let $d(v)$ denote the shortest path distance from the source s to v
  - Thus $d(s) = 0$
  - We will maintain a "label" ℓ_v for each vertex v in V
  - We initialize ℓ_s to 0 and ℓ_v to ∞ for $v \neq s$

Dijkstra's SSSP Algorithm

- We maintain a subset U of V that is initialized to $\{s\}$
- In each of $|V| - 1$ iterations, we add a vertex to U
- We maintain the following key invariants
  - For each vertex u in U, we have $\ell_u = d(u)$
  - For each vertex v in $V \setminus U$, we have

    $$\ell_v = \min_{u \in U : (u,v) \in E} d(u) + w(u, v)$$

    (or ∞ if the minimization is over an empty set); this is an upper bound on $d(v)$
- How do we choose which vertex to add to U in each iteration, and how do we maintain the key invariants?

A General Iteration

- Let u be a minimum-label vertex in $V \setminus U$
  - Observe that any path from s to a vertex in $V \setminus U$ has cost at least ℓ_u; hence $\ell_u \leq d(u)$
  - Since $d(u) \leq \ell_u$ by the second key invariant, we conclude that $\ell_u = d(u)$
- We add u to U
- The first key invariant is maintained since $\ell_u = d(u)$
- To re-establish the second key invariant, we update ℓ_v to

$$\min(\ell_v, d(u) + w(u, v))$$

  for each vertex v in $V \setminus U$ such that $(u, v) \in E$

Efficient Implementation of Dijkstra's Algorithm

- We can use a heap to maintain the labels of the vertices in $V \setminus U$
  - We use $O(|V|)$ Insert and Delete-Min operations
  - We use $O(|E|)$ Decrease-Key operations
- Using an elementary heap data structure, the algorithm runs in $O(|E| \log |V|)$ time
  - Using an array to maintain the labels, we obtain an $O(|V|^2)$ bound, which is an improvement for sufficiently dense graphs
- Using a more sophisticated data structure such as a Fibonacci heap (to be discussed in a later lecture), we obtain a bound of $O(|E| + |V| \log |V|)$

The Minimum Spanning Tree Problem

- We are given a connected, undirected graph $G = (V, E)$ where each edge e in E has an associated weight $w(e)$ (which may be negative)
- A (graph-theoretic) tree is a graph that is acyclic and connected
- A spanning tree T of G is a subgraph $G' = (V, E')$ of G that is a tree
  - It is convenient to identify T with its edge set
  - It is easy to prove that all spanning trees of G have cardinality $|V| - 1$
- The weight of a spanning tree T is defined as $\sum_{e \in T} w(e)$
- A minimum spanning tree (MST) of G is a spanning tree of G of minimum weight

Key Observation

- Let e be a minimum-weight edge in E
- Claim 1: Some MST of G includes e
- To prove this claim, we can use an exchange argument
  - Let T be an MST of G that does not include e
  - If we add e to T we get a unique cycle C
  - For any edge e' on C, $T + e - e'$ is a spanning tree of G with weight at most that of T, and hence is an MST of G
- We can obtain an MST of G by recursively computing an MST T' of the graph G' obtained by "contracting" edge e, and returning $T' + e$
  - Alternatively, we can get an iterative implementation by repeatedly selecting a minimum-weight edge that does not form a cycle with any subset of the previously selected edges

Kruskal's MST Algorithm

- Index the edges $e_1, \ldots, e_{|E|}$ in nondecreasing order of weight
- Initialize T to $\emptyset$
- For i running from 1 to $|E|$
  - If $T + e_i$ is acyclic, then add e_i to T
- Return T

Correctness of Kruskal's Algorithm

- We first claim that the output T is a spanning tree of G
- Clearly, T is acyclic
- Suppose (V, T) has connected components $G_1, \ldots, G_k$ where $k > 1$
- Since G is connected, there is an edge e in E such that the two endpoints of e belong to distinct components G_i and G_j
- But then $T + e$ is acyclic, a contradiction

Correctness of Kruskal's Algorithm (cont'd)

- It remains to prove that T is an MST of G
- Let T_0 be an MST of G
- If $T = T_0$, we are done, so assume $T \neq T_0$
  - Let i be the least integer such that $e_i \in T \oplus T_0$
  - Observe that $e_i \in T \setminus T_0$
  - Let C denote the unique cycle in $T_0 + e_i$
  - Let e_j denote an edge in $C \cap (T_0 \setminus T)$; thus $i < j$ and $w(e_i) \leq w(e_j)$
  - Since T_0 is an MST and $T_1 = T_0 + e_i - e_j$ is a spanning tree, we have $w(e_j) \leq w(e_i)$
  - Thus $w(e_i) = w(e_j)$ and T_1 is an MST with $|T \cap T_1| = |T \cap T_0| + 1$

Correctness of Kruskal's Algorithm (cont'd)

- If $T_1 \neq T$, we can repeat the previous argument with T_1 playing the role of T_0 to obtain an MST T_2 such that $|T \cap T_2| = |T \cap T_0| + 2$
- Let k denote $|V| - 1 - |T \cap T_0|$
- After k iterations, we obtain a sequence of $k + 1$ MSTs $T_0, \ldots, T_k$ such that $T = T_k$
  - Thus T is an MST
- All of the T_i's have the same distribution of edge weights
  - That is, for any given weight z, all of the T_i's contain the same number of edges of weight z
  - Since T_0 is an arbitrary MST of G, we conclude that all MSTs of G have the same distribution of edge weights

Some Other Properties of MSTs

- If all of the edge weights are distinct, there is a unique MST
  - Follows from the preceding claim that all MSTs have the same distribution of edge weights
- Kruskal's algorithm can generate any MST
  - When indexing the edges, Kruskal's algorithm can break ties arbitrarily
  - To produce MST T, favor edges in T over edges not in T
- The set of MSTs does not change if we replace each weight x with $f(x)$ for some increasing function f
  - This transformation preserves the relative order of the weights, which is all that Kruskal's algorithm looks at

Efficient Implementation of Kruskal's Algorithm

- Adding an edge e to T creates a cycle if and only if the two endpoints of e belong to the same connected component of (V, T)
- If $T + e$ is acyclic, so that we add e to T, then the two connected components bridged by e are merged into one
- We can use a "union-find" data structure to manage the vertex sets associated with the connected components of (V, T)

Union-Find

- A union-find data structure maintains a collection of disjoint sets, subject to the following operations
- $\textsc{Make-Set}(x)$ forms a new singleton set $\{x\}$
  - To maintain disjointness, we require that x does not belong to any of the existing sets in the collection
- $\textsc{Union}(x, y)$ merges the set containing x with the set containing y, where x and y belong to distinct sets
- $\textsc{Find-Set}(x)$ returns the "name" of the set containing x (x is required to belong to some set)
  - We require that $\textsc{Find-Set}(x) = \textsc{Find-Set}(y)$ if and only if x and y belong to the same set
- We will study a fast union-find data structure in a later lecture

A Union-Find Implementation of Kruskal's Algorithm

- At the outset, we perform a $\textsc{Make-Set}(v)$ operation for each v in V
  - Thus our initial sets are the vertex sets of the $|V|$ connected components of $(V, \emptyset)$
- For an edge $e = (u, v)$, we check whether $T + e$ contains a cycle by asking whether $\textsc{Find-Set}(u) = \textsc{Find-Set}(v)$
  - We perform $2|E|$ $\textsc{Find-Set}$ operations
- When we add an edge $e = (u, v)$ to T, we perform a $\textsc{Union}(u, v)$ operation
  - We perform $|V| - 1$ $\textsc{Union}$ operations